Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 223–245
Science of Computer Programming
www.elsevier.com/locate/scico

Using role-based coordination to achieve software adaptability

Alan Colman*, Jun Han

Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Victoria, Australia

Received 15 September 2005; received in revised form 15 March 2006; accepted 15 June 2006
Available online 20 September 2006

Abstract

Software systems are becoming more open, distributed, pervasive, and connected. In such systems, the relationships between loosely-coupled application elements become non-deterministic. Coordination can be viewed as a way of making such loosely coupled systems more adaptable. In this paper we show how coordination-systems, which are analogous to nervous systems, can be defined independently from the functional systems they regulate. Such coordination-systems are a network of organisers and contracts. We elaborate how contracts can be used to monitor, regulate and configure the interactions between clusters of software entities called roles. Abstract management contracts regulate the flow of control through the roles and provide monitoring interception points. Concrete contracts are domain specific and allow the specification of performance conditions. These contracts bind clusters of roles into self-managed composites — each composite with its own organiser role. The organiser roles can control, create, abrogate and reassign contracts. Adaptive systems are built from a recursive structure of such self-managed composites. A prototype framework has been built from which adaptive applications can be derived. This framework uses *association-aspects* as a mechanism to implement contracts.

Keywords: Adaptive software; Contracts; Roles; Software organisations

1. Introduction

Software systems are becoming inexorably more open, distributed, pervasive, mobile and connected. This accelerating trend is being driven by new technologies that take advantage of the rapidly increasing power and falling cost of hardware and networking. Consequently, the environments in which software applications operate are becoming more diverse and dynamic. Applications will need to be able to adapt to these changing environments. Furthermore, applications may also have to adapt to changing quality requirements related to the goals of various users.

One basic approach to building runtime adaptive systems is to construct them of loosely coupled elements. These elements are dynamically coordinated and reconfigured to meet environmental demands or changing goals. This underlying approach is common to autonomic computing, agent systems and dynamic architectures. This paper shows how coordination functions can be implemented as a *separate* subsystem that manages *heterogeneous* elements

* Corresponding author.
E-mail addresses: acolman@swin.edu.au (A. Colman), jhan@swin.edu.au (J. Han).

0167-6423/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2006.06.006

(components, agents, services, etc.), and that can be *distributed* in a consistent manner throughout the system at different levels of granularity.

This coordination-system can be described and controlled independently from the functional subsystems that interact directly with the application domain. This approach is analogous to the coordination-systems that exist both in living things and in man-made organisations. In the realm of biology, the nervous system can be viewed as a system that, in part, coordinates the respiratory, circulatory, and digestive systems. Similarly, the management structure or financial system in a manufacturing business can also be described at a separate level of abstraction from the functional processes that transform labour and material into products. Coordination-systems maintain some form of representation of the *requirements* and *current state* of the underlying functional system. These models will vary depending on the variables that need to be controlled in order to maintain the system's viability (ability to survive and fulfil its function) in its environment. In terms of a biological analogy, a controlled variable is the level of oxygen supply to the cells. In a business, the variable might be the amount of funds in the bank. In computerised systems, such control variables could be derived from utility functions that measure computational or communication resources; or variables in the environment with which the system interacts.

This paper is structured as follows: Section 2 describes a motivating example for adaptive role-based systems. Section 3 gives an overview of our Role-Oriented Adaptive Design (ROAD) approach for coordination systems based on roles. Section 4 examines *contracts* between roles. It defines the properties of a ROAD contract, then shows how the control of interaction in a contract can be abstracted from its functional properties. This abstract management level also enables us to define performance measurement points for various synchronisation approaches. We then elaborate the general properties of application specific concrete contracts that inherit from these abstract management contracts. In Section 5 we discuss how self-managed composites can be created from roles, players, contract and organisers. We define the properties of organiser roles and their players that monitor and control such self-managed composites. We also show how adaptive coordination systems can be built from a network of organiser roles and the contracts they control. Section 6 discusses the implementation of the ROAD framework, illustrates how application specific coordination systems can be built using this framework, and, in particular, focuses on how contracts can be implemented using association-aspects. After looking at related work in Section 7, we conclude in Section 8.

2. Motivation

To illustrate how role-based coordination can be used to create adaptive software systems, we will use as an example a mixed-initiative [21] automated manufacturing production system. Let us consider a highly simplified manufacturing department that makes widgets. This department has an organisational structure consisting of a number of different roles that perform different functions (the rounded rectangles in Fig. 1). These roles are Foreman, ThingyMaker, DooverMaker and Assembler (who assembles thingies and doovers into widgets). The Foreman's role is to supervise ThingyMakers, DooverMakers and Assemblers and to allocate work to them. The WidgetDepartment also has a manager role (an *organiser*) that is responsible for creating the associations between the various functional roles, and for assigning entities to play those roles. The WidgetDepartment is, therefore, a composite of these roles and the players who perform them. This WidgetDepartment itself plays the WidgetMaking role within the broader Manufacturing Division of the Company.

These roles can be performed by a variety of heterogeneous players. Players can be software objects, components, agents, external services, machine controllers, or humans interacting with the application through a user interface. Or players can themselves be composites of roles and players. In our example, the thingies are produced by machines that interact with the application through controller components; doovers are outsourced from an external supplier via a Web Service interface; assembler players are interactive UI components that require human employees who do the physical assembly to record their work; and the foreman is an automatic work scheduling component that has been provided by the Company's legacy scheduling software.

Players will vary in their *capability* to perform a role. Capability has both *functional* and *non-functional aspects*. To be able to perform the role at all, the player must meet the functional requirements of the role, that is, be able to perform the tasks allocated to the role. But the player will also have to meet non-functional requirements such as speed, accuracy, reliability and so on. In conventional object-oriented design, all objects of the same type are treated as having identical capability and behaviour. However, in a more *open* system such as our WidgetDepartment, we cannot assume that all players of, say, the role ThingyMaker will have the same capabilities. For example, some types

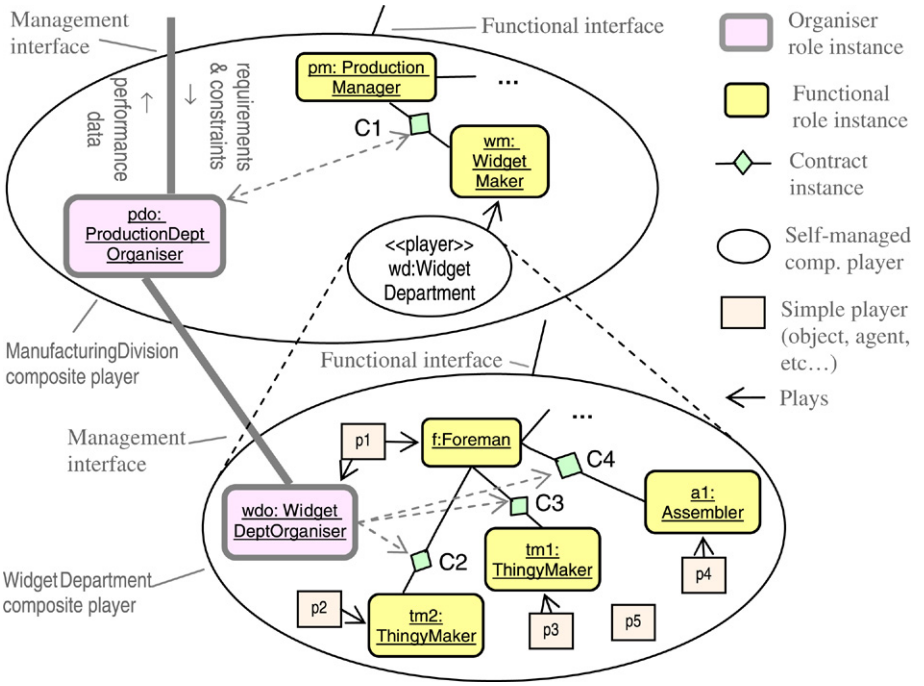

Fig. 1. Exploded view of nested self-managed composites (not all roles and players shown).

of ThingyMaker machine may be faster, more accurate, or more costly than others. Even if the machine-players are of the same type, instances of that type may be running in different contexts that affect their relative performance with respect to their roles. A player may be better connected, better resourced, better maintained, available more often, etc. than other players of the same type.

Adaptive systems must respond to changes in the requirements of the system, or changes in the environment, or both. The WidgetDepartment needs to be able to cope with both changes in the demand for its widgets, and with changes in the capabilities of its players. For example, orders flowing into the department to make new widgets might increase, such that they exceed the capacity of the department to manufacture them. Or, one of the ThingyMaker players may become unavailable, unreliable or too costly. It follows that there must be a representation of the non-functional requirements of the roles in the WidgetDepartment, and a way of measuring whether or not the players of those roles are meeting those requirements.

As a role-based organisation structure, the WidgetDepartment has two basic adaptive strategies available. Firstly, it can restructure the relationships between the roles in the Department. For example, an additional ThingyMaker role (and player) could be added to cope with an increase in demand. The second strategy is to replace a player with an alternative player that better matches the required capability. For example, the production of thingy parts might be outsourced to a third-party service. The appropriateness of these adaptation strategies needs to be evaluated in order to mitigate any gap between required and actual performance.[1] There must also be a means of enacting that strategy.

If an adaptive organisation is to be restructured or have new players added, there must be a way of ensuring that interactions between the roles are appropriate to the organisational requirements, and that the players in the organisation are well-behaved. This is particularly the case as systems become more open, as the components, agents or services that play roles are not necessarily well-tested or well-trusted. Interactions between roles (as player proxies) therefore need to be actively controlled.

In summary, there are a number of capabilities that a role-based system needs if it is to be adaptive at runtime to both changes in requirements and to changes in the operating environment. These capabilities include the ability to *represent* requirements; to *measure* the performance relative to those requirements; to *evaluate* strategies for adaptation; to *restructure* relationships between roles; to *select* appropriate players for those roles; and the ability to

[1] We use the word ‘performance’ in this paper in a very general sense, i.e. the actual level of fulfilment of any non-functional requirement.

control the interactions between players via those roles. This paper introduces an architectural framework for building software systems with such capabilities.

3. Approach

The ROAD (role-oriented adaptive design) approach to designing adaptive software systems is based on an organisational perspective of such systems, i.e., it views a software system as an *organisation* consisting of inter-related *roles*. In the system, these *organisational roles* are played by *players* (objects, components, services, agents, humans, and composites). In fact, these roles are first-class runtime entities that can be played by various players at different times. Roles, as performed by their players, satisfy their responsibilities to the system as a whole. In general, *organisational* roles define an abstract function or a 'position' within an organisation [29], while role-players 'do the work'. This is in clear contrast to the concept of roles in object-oriented modelling, where a role is a descriptor for one end of a relationship. In [13] we examine in more depth the relationship between roles and players.

The ROAD approach distinguishes two types of organisational role, namely *functional*-roles and *organiser*-roles. *Functional roles* (or more properly *domain*-function roles) are focused on first-order goals, that is, on achieving the desired application-domain output. Functional roles and their players constitute the *process* as opposed to the *control* of the system. In ROAD these functional roles are decoupled; they do not have fixed references to each other. Functional roles are associated to one another by *contracts* that mediate the interactions between them. The creation and monitoring of these contracts is the responsibility of *organiser-role*s. An organiser-role is also responsible for controlling the bindings between the functional roles and the players, within the composite under its control.

In ROAD, the binary contracts between roles perform a number of functions Contracts define what functional interactions can occur between the role players. Contracts also define the non-functional requirements of each of the parties with respect to those interactions; and they measure the role-players' performances against those requirements. These contracts are in turn controlled by the organiser of the composite. The organiser-role holds a reflective representation of its composite along with mechanisms for reconfiguring its structure by creating and changing roles, contracts and role-player bindings. The organiser-role player has the adaptive strategies necessary to adapt the composite to changing requirements and environments.

Each organiser role is responsible for a cluster of functional roles. We will call these regulated clusters of roles 'self-managed composites' because each composite attempts to maintain a homeostatic relationship with its environment and other composites. We use the word *composite* rather than *component* because the players that perform roles in the composite are not necessarily encapsulated in a package, but could, for example be an external service. A role-based organisation is built from a recursive structure of self-managed composites. Such a recursive structure is illustrated in Fig. 1.

The self-managed composite WidgetDepartment plays the role WidgetMaker in its 'enclosing' Manufacturing-Division composite. The functional and non-functional requirements of the WidgetMaker role are defined in contract C1. The WidgetDepartment's organiser-role (wdo), who is played by p1, creates, monitors and controls the contracts (C2, C3, ...) between functional roles (f, tm1, tm2, a1, ...), and binds players (p1, p2, p3, ...) to those roles. This structure is coordinated through a network that connects the organiser roles of each of the composites. The network of organiser roles and the contracts they control constitute the *coordination-system* (note that Fig. 1 contains further details that will be referred to in later sections).

In the following sections we show how a coordination-system, built from contracts and organisers, can be imposed on a cluster of functional roles to enable the system to better cope with uncertain environments and changing non-functional requirements.

4. Contracts between roles

In the first part of this section we summarise our previous work [10] on using contracts to manage interactions between roles and give a general definition of what we mean by a 'contract'. We then distinguish abstract management contracts from concrete functional contracts.

4.1. A definition of contracts

ROAD contracts [10] are binary association classes that express the obligations of the contract parties to each other. They define the functional interactions that can occur between the role players, define the non-functional requirements of each of the parties with respect to those interactions, and measure the role-player's performances against those requirements. ROAD contracts can be characterised by the following features (illustrated in Fig. 3):

- The names of the parties to the contract. A ROAD contract binds roles of particular types (e.g. Foreman, ThingyMaker).

Contracts will also have a number of clauses. Clauses can be of three types: terms; general clauses; and protocol clauses:

- The *terms* of a contract are clauses that specify what one party can ask of the other party. The collection of terms defines the parties' mutual functional obligations. For example, a contract term may specify that a ThingyMaker is obliged to fulfil requests to make thingies from its Foreman. Non-functional attributes (utilities) are associated with those terms — for example, the minimum performance standard, the price, quality of service etc. Each term of the contract can have one or more agreed utility functions that measure this performance. Contracts may also contain provisions that define remedies if a clause is breached, or if there is underperformance, by one of the parties. Some terms may 'go to the heart of the contract' in which case breach of a clause leads to termination of the contract.
- *General clauses* in a contract define the preconditions for the contract's instantiation. These include any conditions relating to commencement, continuation, and termination of the contract.
- *Protocol clauses* define sequences of terms to be followed by the parties [19,31]. For example, a foreman might be required to allocate the resources (thingy parts) to a thingyMaker, before it can ask the thingyMaker to make a thingy. A number of existing approaches exist for specifying interaction protocols [7]. While we will not discuss protocol clauses in this paper any further, we envisage extending the ROAD framework by using one such existing approach.

As well as having the above attributes, ROAD contracts have a number of incarnations: *general form* (*à la* class), *specific contract* (*à la* object) and an *execution state*.

- The *general form* (type) of a contract sets out the mutual obligations and interactions between parties of particular classes (e.g. Foreman, ThingyMaker). Clauses applicable to all contracts of that type can be defined. Such clauses may express interaction patterns (Party A will do X when Party B has done Y). Clauses may themselves be the subject of interaction patterns (Clause 2 will take effect after Clause 1 is fulfilled).
- A *specific* contract puts values against the variables in the contract *schedule* (e.g. Foreman and ThingyMaker are named, date of commencement agreed, performance conditions put on clauses etc.). Extra clauses, not in the general form of contract, may also be added.
- A contract is *instantiated* with an identity when the concrete contract is 'signed' – that is, when specific parties are bound to the contract.
- The terms of a contract have *execution states*. Each term has a state machine that maintains the state of interaction between the parties with respect to that term (e.g. Party A has asked Party B to do α under the terms of Term X, but Party B has not yet complied). We call the interaction that occurs during the execution of a term, a *transaction*. Contracts also have an execution state. Contracts states can include incipient, active, suspended, breached, terminated and discharged.

In ROAD, contracts between roles are currently binary. While there is no technical impediment to creating roles with more than two parties, if a number of parties are jointly responsible for the performance of a contract term it can be difficult to assign responsibility in the event of failure. As one purpose of ROAD contracts is to identify underperforming role players, making all contracts binary simplifies this process. The limitation of binary contracts is that they cannot model complex interdependencies between three or more parties. In ROAD, interdependencies between contracts are handled by the composite's organiser (see Section 5.2 below).

Contract Type Foreman–ThingyMaker Contract	
Contract Instance Name	*ft1*
Parties Party A: Foreman	*f1*
Party B: ThingyMaker	*tm1*
Terms	
1. B *must* make thingies on request from A	*NFR1 Moving average >= 5 thingies/minute*
PRE: qty (thingyparts) > 0	*NFR2: cost* = $*1/thingy*
2. A *may* provide thingy parts to B	*NFR: none*
Breach conditions	*B provides < 2 thingies/minute*
General Clauses	*contract state is suspended if B under maintenance*
Current state	*incipient*

Fig. 2. Example of a contract instance between a Foreman and a ThingyMaker.

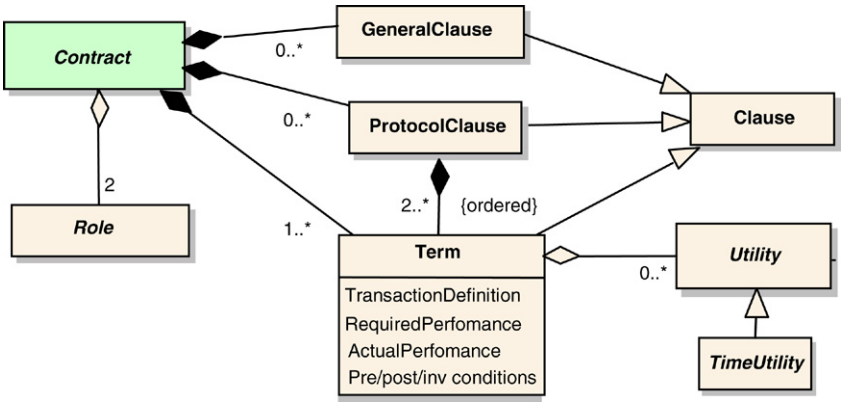

Fig. 3. ROAD contracts — basic concepts.

Fig. 2 is an example of a (partial) Foreman–ThingyMaker Contract. The specific schedule values of the contract instance are in italics. As described in Section 6, such contracts are implemented by extending the Java classes of the ROAD framework.

Fig. 3 shows the basic concepts that make a ROAD contract, and that serve as a meta-model for the implementation of contracts in the ROAD framework. The elements in the diagram are discussed in more detail in subsequent sections.

Real-world commercial contracts are passive bits of paper that are monitored, and to some extent enforced, by the parties. A ROAD software contract, on the other hand, can store dynamic state-of-execution information in the contract itself. It can also enforce the terms of the contract by controlling the interactions between the parties.

4.2. Contract abstraction

Contracts between functional roles often share common characteristics. In particular, the *control* aspects of the contract can be abstracted. These control relationships between roles in the structure can be viewed as an *organisational* description of the system. Such organisational descriptions are based on the conceptual separation of control from process — the separation of management of the process from the process itself. Management functions can, to some extent, be characterised in a domain-independent way. These functions include coordination, goal-transmission, regulation, accounting, resource-allocation, auditing and reporting. The organisational perspective is one of a number of possible perspectives, but it is a perspective that allows us to explicitly represent and incorporate adaptive mechanisms into a system.

Using our example of a WidgetDepartment, the control-management relationship between a Foreman and an Assembler could be characterised as a Supervisor–Subordinate relationship. Similarly, a Foreman–ThingyMaker relationship would also be characterised as a Supervisor–Subordinate relationship. Abstract *operational-management* roles, such as a *supervisor* and a *subordinate*, are association-end roles. They do *not* define a position in an organisational structure as do functional roles. Rules can be defined that control the interactions between such *operational-management* roles. For example, a supervisor can tell a subordinate to do a work related task but a subordinate *cannot* tell a supervisor what to do. At the programming level, this means that subordinate

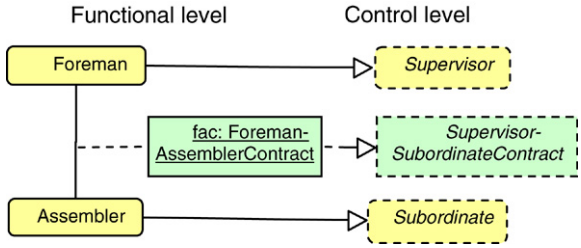

Fig. 4. Functional and management contracts.

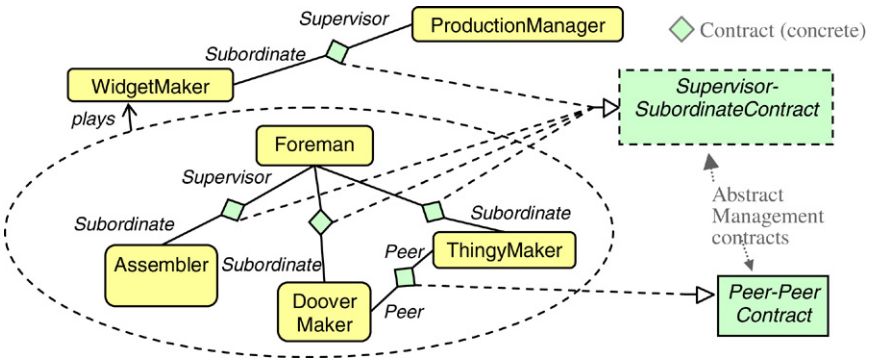

Fig. 5. Domain specific abstract organisation bound by contracts.

roles cannot invoke particular categories of methods in supervisor roles. Other types of management contract include auditor–auditee; peer–peer; and predecessor–successor in supply-chain and production-lines.

In ROAD, these abstracted control aspects of the relationships between roles are encapsulated in an abstract *management contract*. Concrete contracts inherit control relationships from these management contracts. The conceptual relationships between concrete and abstract management contracts, and the respective roles they associate, are illustrated in Fig. 4.

As can been seen from Fig. 4, fac is an *instance* of a ForemanAssemblerContract which is an association class between the Foreman and the Assembler roles. The ForemanAssemblerContract inherits the form of its control relationships from the SupervisorSubordinateContract by virtue of the fact that the Foreman has a Supervisor role in relation to the Assembler's Subordinate role.

Fig. 5 illustrates an organisational structure for our WidgetDepartment. This structure is similar to the Bureaucracy pattern [32] but is built using contracts. In order to simplify the diagram, contracts have been drawn as diamonds. The structure, which is similar to a business organisation chart, is still abstract because players have not yet been assigned to roles. Note that the Foreman *functional* role has *both* Supervisor and Subordinate *operational-management* roles in the organisational structure relative to other functional roles with which it interacts.

4.3. Abstract contracts at the management level

A contract *term* defines a transaction that expresses the obligation of one party to the other party. The communications that occurs between the parties in such a transaction can be viewed, at an abstract level, as expressing control relationships. Communications between the parties can be represented as abstract control messages. This allows us to characterise three properties of contracts and terms: the authority relationship between parties to the contract; the sequence of abstract messages in a term; and the *points* at which the performance (or non-performance) of the contract term can be measured for various types of transaction. These three properties of abstract management contracts are examined in the next three sections.

Expressing authority using abstract message types

As software systems become more open, dynamic and complex, the developer of an application does not necessarily have full control over all the components, services or agents that make up the application. In ROAD, all application-related interactions between such players occurs via the contracts that associate their roles in the organisational

Table 1
Example set of abstract message types represented by CCA primitives

D	DO	Q	QUERY	R	RESOURCE_REQUEST
G	SETGOAL	I	INFORM	A	RESOURCE_ALLOCATE
Y	ACCEPT	N	REJECT		

structure. Contracts *restrict* the types of method that particular role instances can invoke in each other, thus ensuring that role-players are well-behaved with respect to the application. While such restrictions can always be written into domain-specific concrete contracts, there are a number of common patterns of authority relationship that can be generalised into abstract management contracts. From our example above, the Supervisor–Subordinate operational-management role association restricts interactions between the entities playing the ThingyMaker and the Foreman to certain *types* of interaction. For example, a ThingyMaker cannot invoke actions in a Foreman-Supervisor. Furthermore, these contracts only *allow* interactions between particular instances of role-players. For example, the method ThingyMaker.setProductionTarget() can only be invoked by the ThingyMaker's own Foreman. A Foreman that is not contracted to the ThingyMaker could not invoke the method.

The control communication in management contracts can be defined in terms of *control-communication act* (CCA) primitives. These performatives abstract the *control* aspects of the communication from the functional aspects. In [10], we defined a simple set of abstract CCAs for direct and indirect (resource constraint) control and for information passing. Table 1 defines an example set of such abstract message primitives suitable to a hierarchical organisation.

Unlike the communication act primitives in agent communication languages, such as FIPA-ACL [17], CCAs express only *control* relationships between the two parties bound in a contract, rather than the types of communication between two independent agents. Expanding our example of a Supervisor–Subordinate contract using the primitives we defined above, Supervisors can initiate some types of interaction and Subordinates can initiate others. Initial CCAs for these roles are:

Supervisor initiated: DO, SET_GOAL, INFORM, QUERY, RESOURCE_ALLOCATE
Subordinate initiated: INFORM, QUERY, RESOURCE_REQUEST

Other forms of management contract, such as Peer–Peer, would have different sets of valid initial CCAs for each party. For example, peers might be able to initiate messages corresponding to all the above CCA types, but the respondent peer has the option of replying with a REJECT. We discuss sequences of CCAs in the following section.

If needed, the set in Table 1 could be extended to capture a referential command relationship (A tells B to tell C to do something). Alternatively, the set could be expanded to express propose-commit type communications that might be found in agent communication, or in a database two-phase commit. However, while the above set is only a basic set of CCAs, it is sufficient to allow us to define a number of contracts between operational-management roles. From these contracts we can create organisational structures such as those in Fig. 5.

Contract transactions as CCA sequences

As well as controlling individual method invocations, contracted transactions often involve a sequence of interactions. The contract needs to track these interactions to ensure that contractual obligations are being followed by the parties, and to know when a contractual transaction is completed. A transaction instance performed under a term of the contract can be viewed as a sequence of CCAs. These CCAs are abstractions of the actual underlying messages or method invocations. These sequences are at the same level of abstraction as CCAs. Only the form of communication between the parties is represented. There is no information about the content of the task. There are a limited number of these sequences that form sensible interactions. For example, both QUERY → INFORM, and RESOURCE_REQUEST → RESOURCE_ALLOCATE make sensible CCA sequences between an initiator and respondent, whereas DO → RESOURCE_ALLOCATE (presumably) does not make sense.

What constitutes a valid sequence also depends on the type of request synchronisation used in the transaction. The CORBA middleware standard defines four approaches to request synchronisation: Oneway, Synchronous, Asynchronous and Deferred-Synchronous [16]. Similar distinctions between synchronisation approaches are needed in the definition of valid CCA sequences. For example, a Oneway or Synchronous request does not require any separate response from the respondent. In these circumstances a single DO type invocation may be a valid transaction. On the other hand, if an Asynchronous approach is implemented a reply would be expected in the form of, for example, a DO-INFORM sequence.

Table 2
Example form of an operational-management contract

Management contract	
Name	`Supervisor-Subordinate`
Party A	`Supervisor`
Party B	`Subordinate`
A initiated terms	
– Oneway	`I; G; A`
– Synchronous	`D; Q`
– Asynchronous	`Dy; Di; Gy; Gi; Qi`
– Deferred synchronous	`DQ, GQ`
B initiated terms	
– Oneway	`I`
– Synchronous	`Q; R`
– Asynchronous	`Qi; Qn ; Ra; Rn ; Ri`

It is possible to represent these sequences as regular expressions made up of CCAs between initiator and respondent. To do this we will encode the CCAs as single letters (as in Table 1) so that complete transactions can be represented as individual strings. For further expressiveness, we can apply the convention that initiator CCAs are capitalised, and respondent CCAs are in lower case. For example, the string "Dy" indicates that the initiating party asks the respondent to do something, and that the respondent subsequently accepts. A Deferred-Synchronous DO sequence (where the initiator is responsible for getting the result of the transaction) could be expressed as "DQ". Other sequences, such as a deferred-asynchronous ("DyQi"), could also be defined if needed.

The form of our Supervisor–Subordinate management contract has been summarised in Table 2. The valid protocol sequences are expressed as strings as defined above. The contract below defines terms that permit asynchronous and deferred-synchronous interaction. However, Oneway DOs are not permitted as we want to measure the time-performance of action requests, and to do this we need a response when the transaction is complete. The terms of the contract will be violated if the sequence of interactions does not follow one of these strings. Other types of management contract will have different sets of permissible sequences. For example, the protocol "DiN" is presumably acceptable in a Peer–Peer management contract, where the "i" is a conditional accept and the initiator rejects the condition.

We can further extend the CCA sequences to take account of a non-response to a message where, say, communication channels are unreliable. For example, D*xi would express the situation where the initiator can send a DO type message up to x times before there is a violation of the contract term. For each transaction where a response is expected, values for the response timeout and the number of permissible retries (x) would be specified (this is done at the concrete level of the contract).

If contracts are to enforce CCA sequences, they need to keep track of the *state* of communication between the roles that are party to the contract. This implies that there must be an instance of a contract for every association between roles. For machine processing, control abstractions of transactions can be represented by finite state machines (FSMs). These FSMs keep track of the transactions between two contracted parties and report violations. Clauses can have as a goal the *maintenance* of a state or the *achievement* of a state. In the case of maintenance clauses, successful transactions of the protocol will result in a return to a 'ready' state. The successful completion of achievement clauses will result in a 'completed' state for that clause. The nodes in Fig. 6 represent CCAs issued by either the initiator or the respondent in the transaction of a particular clause in the contract. The letters in the nodes are a shorthand for CCAs, as defined Table 1 (initiator CCAs in capitals, respondent CCAs in lower case). The FSM for the Supervisor–Subordinate contract is illustrated Fig. 6. It shows *valid* synchronous, asynchronous and deferred-synchronous transactions initiated with a Supervisor DO (for simplicity only timeouts on the DO has been illustrated).

Using CCAs to define performance measurement points

In an adaptive system, we need to know if the role-players are performing their role(s) according to the non-functional requirements (NFRs) defined in the role's contracts, so that adaptive action can be taken if underperformance is detected. The NFRs are defined *with respect to* functional transactions (e.g. *Functional transaction* Task A will be completed in *NFR* Time t). CCAs allow us to define transaction patterns in a domain-

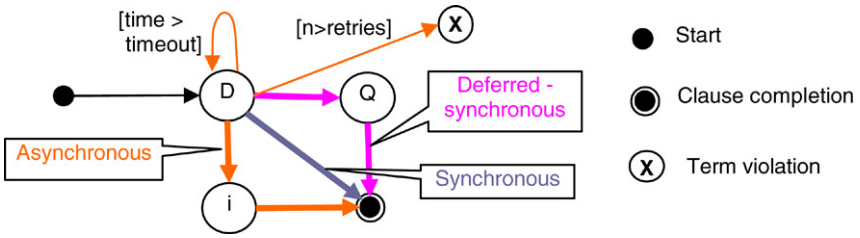

Fig. 6. Valid synchronous, asynchronous and deferred-synchronous CCA sequences initiated with a Supervisor DO CCA.

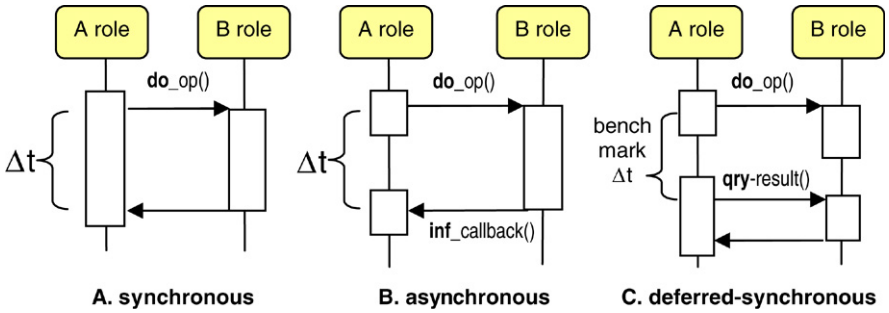

Fig. 7. CCA points intercepted by contract for measurement of time-based performance vary depending on the contract-term's synchronisation type.

independent way. The performance, or non-performance, of an obligated party bound by a contract term can be determined by measuring the change in various states at the start and at the end of the transaction. By intercepting messages between roles, and monitoring the CCAs of those messages, a management contract can determine the start and finish of transaction patterns defined by contract terms. The change of state measured can be of two types: time-dependent and domain-state-dependent. Time-dependent performance is calculated by measuring the time it takes to perform the transaction itself. Calculating domain-state-dependent performance involves measuring some state of the domain or environment before and after the transaction, and then determining the effect of the transaction (e.g. a cost-function is evaluated, or a control-variable is measured.) In ROAD, performance is measured by an arbitrary utility function associated with the contract-term, as shown in Fig. 3.

The type of performance that can be measured, and how CCAs indicate the start and finish of the transaction, will depend on the synchronisation method of the transaction; that is, whether the contract term is oneway, synchronous, asynchronous or deferred synchronous.

Timed performance may need to be measured when the term invokes some action in the obligated party; that is, when a DO or SETGOAL type message is sent. Fig. 7 shows the interception points at which performance is measured for the differing synchronisation methods. In synchronous interaction, time is sampled at the point of method invocation and at the point of return of the method. Asynchronous interaction relies on detecting the INF CCA reply that the obligated party sends when the task is finished. The execution time of a deferred-synchronous interaction cannot be measured directly from interactions intercepted by the contract between roles. This is because there is no message sent on completion of the task from the obligated role to the invoking role. However, the performance of this type of interaction can be measured against a benchmark: the time between the invocation and when the query for results is sent from the invoking role. The obligated party either meets the benchmark or does not.

The above diagram is a simplification; it does not show the role-players to whom the calls are delegated, or show the contract that intercepts the messages between the roles. It is this contract that measures the time at the appropriate points, and calculates the time-based metric. Also note that performance, as measured between abstract messages, does not indicate success or failure of the task itself, as there is no domain semantics expressed. All that is measured is the time elapsed. This measurement needs to be interpreted by the contract into a level of performance.

Domain-state-dependent performance measures the change of state in environment rather than time elapsed during the transaction. By detecting the CCAs with which a transaction starts and finishes, a contract can be made to evaluate an arbitrary utility function that indicates some change of state within the domain or environment. This utility is defined at the concrete level of the contract. Such measurement of performance can be applied to transactions with

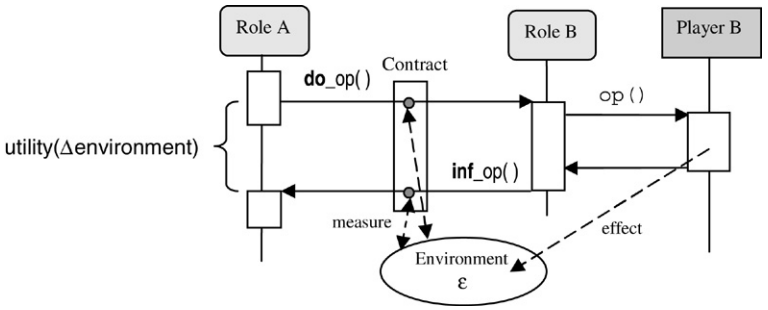

Fig. 8. Change of domain-state measured in an asynchronous transaction.

all types of synchronisation method. Even a oneway interaction, in which no response is received from the obligated party, can be monitored for domain performance provided there is an appropriate delay between the measurement of the initial state and the measurement of the altered state. For example, a classic feed-back control loop is a oneway interaction. The process set by the controller does not return a result — rather a control variable is sampled in the environment to determine the result of the control settings. Fig. 8 shows the measurement of domain-state dependent performance in an asynchronous transaction. Role A invokes some action in Role B. This invocation is intercepted by the contract which measures the environment ε before passing invocation to Role B, resulting in the invocation being enacted by B's player. This action changes the environment. The response message is intercepted by the contract and any change in the environment ε is measured. For example, if Player B is an external service that charges for the provision of a function, the contract could access the accounting system before and after the transaction.

Contracts at the management level are limited in that they only monitor or enforce the *form* of the communication between the parties. Abstract types of interaction may be restricted, and transaction patterns monitored for performance — but there is no domain content apparent at the management control level of abstraction. Domain-specific content is defined at the concrete, functional level of the contract.

4.4. Concrete contracts

A concrete contract type inherits its control patterns from the abstract management contract as described above. For example, the Foreman–ThingyMaker contract inherits the CCA control patterns from the Supervisor–Subordinate abstract management contract, which are applied to the specific methods of the Foreman and ThingyMaker (see below). In addition, as was illustrated in our example of a Foreman–ThingyMaker contract in Section 4.1, an instance of a concrete contract further *specifies* the parties and clauses of the contract. All contracts need to specify the following items:

- *Parties*. The *types* of functional role that can be bound to the contract are specified. In the example of a contract in Fig. 2, only instances of roles of type Foreman and type ThingyMaker can be bound to the contract.
- *Terms* of the contract. Each term defines a transactional obligation of one party to the other. These are expressed as an initial method signature that can be invoked in the obligated party. When a contract specialises an abstract management contract, all functional role method invocations and responses between the parties are associated with CCA primitives. For example, the method do_makeThingy( ) of the ThingyMaker tm would be matched to the DO CCA. This means a supervisor, contracted to tm, can invoke this method. If a CCA sequence is to be enforced for a transaction, a valid CCA *regular expression* (as defined in the management contract) is assigned to the interaction, and values are specified for retries and timeouts. Transactions between the parties must follow any abstract CCA pattern defined in the management level of the contract. If a CCA sequence allows for timeouts, the values for timeouts (in the event of no response), and values for the number of retries that are permitted, are specified. This is done in the concrete contract as these values only make practical sense in relation to a domain specific function.

Optionally, contracts may specify performance:

- *Performance of terms*. The contract defines the required level of performance and measures actual performance. If a term defines an action that must be performed, the contract term can have associated with it a utility function that

measures the obligated party's actual performance. In the example contract (Fig. 2), thingies must be made at a rate > 5 per minute. As set out in the previous section, these utility functions can be time-dependent or domain-state dependent. The actual performance is compared to the required performance to determine the obligated party's level of performance with respect to the term — e.g. performing, underperforming, in-breach. For example, a contract might specify the maximum time allowed for a contracted thingyMaker to produce a thingy. These non-functional requirements (NFRs) reside in, and are measured by, the contract rather than the component itself. As such, the performance requirement of a term can be changed dynamically by the composite organiser.

– *Performance of contracted party*. The purpose of measuring the performance of contracted parties is to attempt to mitigate underperformance, and to replace an underperforming party if necessary. A contract term always has one party that is responsible for its performance. The performance of a party, with respect to the contract, is the aggregation of its performance of such terms for which it is responsible. The significance of underperformance can vary. Some term violations 'go to the heart' of the contract and violation of the critical clause leads to automatic breach of the contract — the contract throws an exception. On the other hand, other terms may not be as critical to the contract and will be monitored. The contract contains metrics to measure the performance of its terms: for example 'average time to make a thingy'. Such metrics are monitored by the organiser role, which may then choose to abrogate the contract if the performance is unacceptable and, for example, if another role-player is available to perform the function. There may also be remedies for clause violation. If a clause is violated it may be permissible to renegotiate the contract to a different service level.

Additionally, conditions for general clauses and terms may be specified:

– *Preconditions, post-conditions and invariants* for the interactions can be specified for both general clauses and terms. General clauses that set preconditions for the contract's instantiation can be defined. These include conditions relating to commencement, continuation, and termination of the contract. In our example, the contract is suspended if the ThingyMaker machine is off-line due to maintenance. Conditions can also be set against the performance of contract terms. These conditions are similar to those defined in the design-by-contract (DBC) approach [28], where such conditions are aimed at ensuring the correct functioning of the software. Consequently, these DBC conditions themselves cannot be changed. In addition to such fixed conditions, ROAD contracts may have conditions that *can* be varied by the organiser (provided the variation does not contravene correctness constraints). This allows variable NFRs to be expressed as conditions of the contract. For example, if there are costs associated with the performance of a function, such as making a thingy, then the contract might specify the acceptable limit of those costs.

During execution, a contract itself monitors the interactions between the roles. The contract will prevent unauthorised or invalid interactions and monitor all transactions in order to maintain the state of execution of its clauses. The contract also keeps the state of any performance metrics updated. If an obligated party underperforms according to a term of the contract, or if a clause is violated, the contract informs the organiser role that controls it. Contracts may also be actively monitored by their organiser roles. We examine the interactions between contracts and organiser roles in the next section.

5. The coordination system

A coordination-system is constructed from contracts and the network of organisers that control them. Every organiser role is responsible for a cluster of functional roles, players and contracts called a *self-managed composite*. In this section we discuss the conceptual relationships between functional roles, self-managed composites, organiser roles and players. We then discuss the function of organisers that manage these composites and distinguish between organiser roles and organiser players. We then show how such a coordination system, in the form of a network of organisers, can achieve adaptive behaviour through the flow of regulatory control messages.

5.1. Structure of an application

A contract, as discussed above, gives us a way to independently and dynamically define the relationship between two loosely coupled functional roles. This late-determination of *indirection of association* allows us to reconfigure the

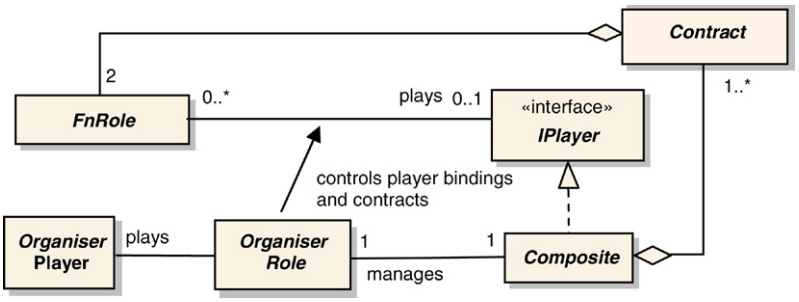

Fig. 9. Conceptual relationship between functional roles, players, and self-managed composites.

organisational structure of the system. Roles are first-class entities but rely on players to execute their functions. A role structure can exist independently of players. A role and its player must be functionally compatible (the player must have the capability of executing the role interface). Roles and players are independent entities. They are dynamically bound to each other by the organiser creating mutual references between them. The binding between a role and a player provides another type of adaptive indirection we call *indirection of instantiation*. Roles can be filled at different times by different players. Players can be transferred depending on the performance requirements of the role and the capability/availability of the players. Roles may be temporarily unassigned to players. A role is a *position* to be filled by a player [29]. For example, if a ThingyMaker machine (player) fails, the role does not cease to exist. That position (role) within the company structure may be temporarily unassigned, but the company may continue to function viably in the short term. Orders can still be taken, current work orders still be manufactured, finished goods can still be shipped, accounts can still be processed and so on. Nor does the identity of the role depend on the identity of the player. From the organisation's point of view it does not matter whether ThingyMaker machine A or B performs the role, or if ThingyMaker production is outsourced as long as the players have the required capability.

If roles can be temporarily unattended at runtime, there must be some form of message-queuing to maintain the viability of the organisation. The following is a minimal definition of a functional role: an interface that expresses the external function of the role, with some sort of mechanism (independent of the players) for queuing messages. If the role is involved in more than one contract, there must also be a mechanism for allocating outgoing messages from the role's player to the appropriate contract as the player is 'blind' to its position in the organisation and only communicates with its role(s). There may also need to be mechanisms for transferring the state of a transaction if players are swapped in the midst of long-lived transactions. In heterogeneous systems, where there is a mismatch in message formats or mechanisms between a player and the role, roles may also function as adaptors. For example, if a role needs to interact with a player that is a Web service, the role will need to convert method invocations to SOAP messages and implement a WSDL interface [11].

As discussed in Section 3, related roles can be organised into clusters called self-managed composites. These composites can form a recursive hierarchy as shown in Fig. 1. If the roles in a self-managed composite are instantiated with players (or capable of being instantiated) then this *composite* can be implemented as a runtime *entity*. At the implementation level, an encapsulated composite is a type of player that can be assigned a role. This relationship between roles, composites and players is illustrated in Fig. 9. The figure also illustrates how the recursive structure of self-managed composites is implemented: a composite contains contracts which bind functional roles which may themselves be implemented (played) by composites.

The elements in the conceptual model can be viewed as abstract classes from which concrete domain-specific classes are implemented. A concrete self-managed composite has two interfaces: a functional interface that connects it with its role, and a management interface that connects its organiser with other organisers.

5.2. Organiser roles and their players

Each self-managed composite has an organiser. The organiser function can be separated into an organiser role that provides the means to manipulate a composite, and a player that controls the reconfiguration. For example, in Fig. 1 the WidgetDepartment wd has an organiser role wdo played by p1 (who also plays the Foreman role). The organiser *role* maintains a *representation* of its self-managed composite and *operations* to manipulate and reconfigure

the structure of its self-managed composite. These operations are standard to all organisers. The operations of an organiser *role* include:

- Create new instances of roles from predefined role types. For example, in Fig. 1, if wdo receives a request for more thingies it can create an new ThingyMaker role tm2.
- Make and break contracts between roles in its self-managed composite. An organiser can instantiate, change, abrogate and reassign contracts between functional roles. e.g. wdo creates the contract c2 between tm2 and the Foreman f.
- Make and break bindings between its roles and players that are available and functionally compatible. e.g. wdo binds player p2 to tm2, or changes tm1's player from p5 to p3.
- Monitor the *actual* performance of contracts in their self-managed composite. This can occur through the contract notifying its organiser of underperformance or breach. Alternatively, the organiser can poll its contracts.
- Update the *required* performance in the terms of a contract.
- Change the *conditions* in both general clauses and terms of a contract.
- Receive non-functional requirements from organisers higher up the coordination system. e.g. wdo receives a request for an increased rate of production from the ProductionDept organiser pdo.
- Transmit non-functional requirements to the organisers of any sub-composites players under its control.

In order to carry out these operations an organiser must maintain a representation of the organisation. It must know:

- What roles and players/sub-composites it is controlling.
- What contract-types it has available to associate those roles.

The above operations and knowledge structures define a generic organiser *role* class. The role defines *how* to manipulate the composite. The responsibility of the player who plays the organiser role is to figure out *what* is to be done. This requires some level of decision making. Separating organiser roles from players allows generic functions to be separated from the domain-specific decision-making process. The decision-making functions that use domain-specific knowledge are implemented in the *player*. While, conceptually, the organiser player is separate from the organiser role, whether or not to implement them as separate runtime entities is a design decision. One advantage of decoupling the organiser role from its player is that players may vary in the intelligence they can apply to the decision-making process. For example, due to changing requirements an organiser player may be unable to reconfigure its composite in a way that fulfils its external obligations. In this case, the organiser can be upgraded to a more intelligent model such as a deliberative agent, human operator or programmer.

An example of a decision making process performed by an organiser is illustrated in Fig. 10. The organiser receives NFRs for the composite from the organiser of the enclosing composite (super-organiser). It needs to translate these composite non-functional requirements (NFRs) into NFRs for terms in the contracts it controls. The organiser also monitors the actual performance of its contracts either actively (polling) or passively (waiting for a notification from the contract). If there is a mismatch between the actual and required performance (either because the requirements in the contract have been changed, or because the actual performance has decreased), the organiser attempts to mitigate this underperformance by reorganising its composite. The organiser chooses a strategy determined by the actual and claimed performance of the existing player and other available players. The *actual*-performance of a role-player pair is the historical performance measured as interactions pass through the contract. If a new player is allocated the role, then this actual performance data must be reset. In the absence of historical performance data, *claimed*-performance information could be obtained from a specification provided by the builder of the player or a third party accreditor.

In Fig. 10, the types of function that *may* need to be implemented in an organiser *player* at the application-domain level (rather than the organiser *role*) have been italicised. These functions required some deliberative capacity. They include:

- Translation of non-functional requirements (NFRs) that are provided by the enclosing composite, to NFRs for contracts within the organiser's composite. For example, a WidgetDeptOrganiser player needs to translate a requirement for widget throughput into NFRs of contract terms related to thingy production.
- Setting of conditions in the form of executable assertions on contract interactions to ensure any ordering constraints involving more than two parties are enforced.
- Ability to discover functional-role players that are candidates to play roles.

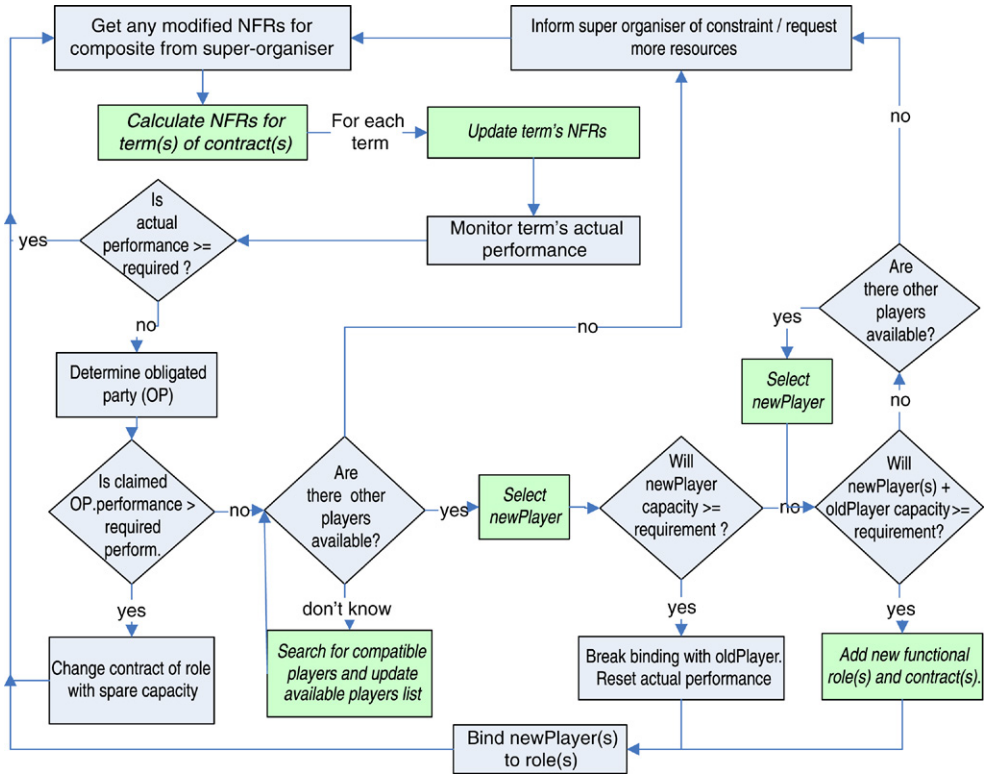

Fig. 10. Example of a decision making process of an organiser.

- Ability to compare candidate player NFR characteristics (claimed and actual performance, availability etc.) and decide on the appropriate functional role-player bindings.
- Ability to decide on appropriate configurations of its composite (i.e. what roles and contracts to instantiate) in order to remediate changing NFRs or changing performance of its players. Any such reorganisation must maintain the composite's viability. For example, if the structure is based on a bureaucracy, the organiser must ensure that proper chains-of-responsibility (i.e. supervisor–subordinate chains) are maintained to preserve the functional flow-of-control.
- If the system is a control system that is dynamically responding to perturbations in the environment, unstable feedback loops can be created. The organiser may need to regulate NFRs to dampen such oscillations.

The extent to which a player needs to implement the above functions will depend on the type of system being implemented. In general, the more *open* the system (that is the more complex the cybernetic *variety* [4] of the composite's environment), the more *capability* the organiser-player will need to maintain the viability of its composite.

5.3. *The coordination system — a network of organisers*

A coordination-system is a hierarchical network in which organisers are the nodes. The structure of this hierarchy reflects the recursive structure of the composites, with the composite representing the overall system at the top. An organiser provides the management interface to its composite and interprets the regulatory control messages that flow through this network into terms for the contracts within its composite. These messages are non-functional requirements expressed in terms of metrics. Each metric is associated with a utility function (as in Fig. 3).

Non-functional performance requirements flow down the coordination hierarchy, and information on the performance of the managed composites (actual or claimed) flows up. If the system uses indirect control to regulate its output, resource constraint information also flows over this link. Information on the projected resource requirements flow up, and permissions to access resources flow down. As defined above, it is the organisers' responsibility to translate these messages into NFRs that are applicable to the contract terms in the various contracts it controls.

We will call these two regulatory control-message flows, respectively, top-down *requirement/constraint-propagation* and bottom-up *performance-monitoring*. We will illustrate their dynamics in the next section.

5.4. Adaptive behaviour

The structure and adaptive behaviour of a coordination-system will be illustrated by looking at a ThingyMaking team within our WidgetDepartment (wd). As shown in Fig. 1, the WidgetDepartment plays the WidgetMaker role within the ManufacturingDivision. The relationship between functional requirements and NFRs is illustrated with a production scheduling problem. We need to keep in mind that in an open system, the time taken to execute a function may vary or come at a cost.

Requirement and constraint propagation. Performance requirements pass down the hierarchy of organiser roles to alter the performance requirements of the contracts. In our ManufacturingDivision the ProductionManager receives (from above) orders for widgets. It determines the priority of the orders, and passes these on to the WidgetMaker role (as determined by the contract C1). The ProductionDeptOrganiser (pdo) receives performance requirements and constraints, and in turn adds derived NFRs to the contract C1. For example, the contract may require that widgets be made within certain time constraints, or that certain resource costs not be exceeded. The WidgetMaker role is played by the WidgetDepartment (wd) self-managed composite. To fulfil its obligations under the C1 contract, the wd must organise the production of thingies and other parts (not shown) within a specified timeframe. For example, the management level of contract C1 allows the ProductionManager pm to invoke the WidgetDepartment do_widgetOrder(. . .)method.

The Foreman f is a *delegate* for the WidgetDepartment's interactions with the WidgetMaker role, and provides the implementation of the composite's functional interface. Via this interface, the Foreman f receives orders for widgets, and in turn allocates work to, among others, the thingyMakers (tm1 and tm2). The Foreman can do this under the terms of the Foreman–ThingyMaker contracts (instances C2 and C3) by invoking ThingyMaker's do_makeThingy() method. While the contracts C2 and C3 have the same form, these instances of the Foreman–ThingyMaker contract have different performance characteristics written into their respective contract schedules. Suppose the role-player attached to tm1 can make 10 thingies per hour, while the role-player attached to tm2 can only make 5 thingies per hour. The Foreman f (party to contracts C2 and C3) can use this performance capacity information in the schedule when deciding to whom the work should be allocated.

Required-performance information on goals and constraints is transmitted through the organiser roles. The organiser role (e.g. wdo) receives NFRs from the organiser above it in the coordination hierarchy (e.g. pdo), then interprets these into performance requirements for the contracts it controls.

Performance monitoring and breach escalation. The organiser role can monitor the contracts to see if they are meeting their performance requirements. Changing requirements, environment or computational contexts can lead to the violation of performance clauses in the contract. If the *actual* performance (as expressed in the contracts) falls below the *required* performance, then the organiser must attempt to reconfigure the composite by altering the existing contracts, reassigning roles to more capable players or creating new contracts and roles. For example, if the combined output of tm1 and tm2 cannot meet the required performance of the composite (as determined by contract C1) then the WidgetDeptOrganiser wdo needs to reconfigure its composite using a strategy similar to that outlined in the flowchart Fig. 10.

If this reconfiguration is beyond the current capability of the composite organiser, then the problem is escalated up to the next level. The organiser at level N must inform the organiser above it (level $N + 1$) in the coordination system hierarchy. For example, in Fig. 1, if wdo detects that the thingyMakers it controls are, or will be, over-loaded, it may ask pdo for resources to get more thingyMakers. Such escalation can be viewed as an organised form of exception-handling, where performance messages flow up through the *coordination*-system before failure occurs in the functional system. Just as an animal detecting a threat will increase its adrenaline levels to stimulate the heart rate for flight or fight, so the coordination-system detects stress on the system then changes the parameters of contracts (or reorganises them) in an attempt to avert system failure.

The performance parameters that an organiser could monitor include the rate of the flow of data resources through the system, the state of communication networks, or the computational loads on the players performing the various roles. The organiser's ability to successfully reconfigure the composite will depend on its ability to sense

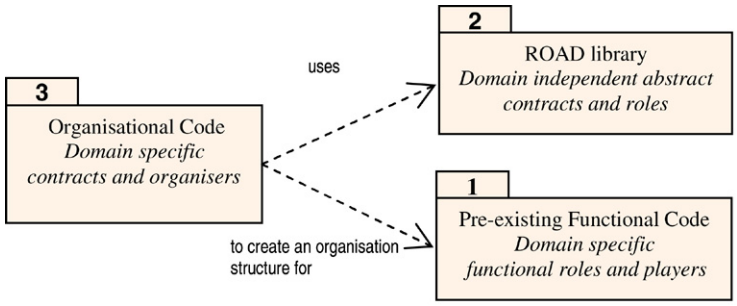

Fig. 11. Organisational code based on the ROAD library package can be written as a separate concern from the functional code.

the performance parameters from either the contracts or the environment, its ability to reason about the causes of underperformance, and its ability to find appropriate strategies to mitigate any underperformance.

Note that the ROAD framework provides a limited form of adaptation we refer to as *ontogenic* adaptation [12]. The framework allows for the reconfiguration of composites and the swapping of role-players in response to changes that lead to differences between required and actual performance. However, organisers in the current framework do not have the ability to compose *new* functions or create new *types* of association. A valid functional composition is assumed as a starting point.

6. Implementing coordination systems

The approach we have taken to implementing coordination systems is to develop a domain-independent framework that implements the concepts described above. The application developer uses this framework to write domain-specific organisational code. A novel feature of this implementation is the use of a type of instantiable aspect called "association-aspect" to implement contracts at the management and functional levels. This section begins by describing the approach taken to implement a coordination system as a separate concern from the functional system. We then give an overview of how association-aspects have been used to implement contracts. We have discussed the binding of role and players elsewhere [13].

6.1. A framework-based implementation

Our implementation of the ROAD framework separates code into three independent packages: domain-specific functional code; domain-independent organisational code; and domain-specific organisational code. Fig. 11 illustrates these three concerns:

1. Domain-specific functional code consists of classes that define the functional roles of the organisation and the entities that can play those roles. Classes representing these roles and players can be written without defining the configuration of the organisation in which they will participate. In ROAD these functional roles are decoupled; they do not directly reference each other. The concrete roles in this package define both required and provided interfaces. These roles inherit message handling capability from the abstract role class defined in the ROAD library. Players implement the interface defined in the concrete roles.
2. Domain-independent organisational code is a reusable library that defines abstract classes for functional roles, organiser roles, composites and utility functions. This library also defines abstract management contracts as described previously (e.g. a Supervisor–Subordinate contract). If the abstract management contracts provided by the ROAD library do not express the necessary control relationships, new abstract contracts can readily be defined as aspects.
3. The third 'package' of code defines the domain-specific organisation. The programmer creates domain-specific contracts between the functional roles defined in the first package. These contracts inherit from abstract management contracts in the ROAD library package. Domain-specific utility classes are also written to extend the abstract utility class. These utility classes allow performance requirements to be specified and enforced for interactions between the particular functional roles that are bound by the contract. Domain-specific organisers are also created to implement specific adaptation strategies.

The code from these three packages is compiled using a modified AspectJ 'association aspect' compiler [33] to create an adaptive application. The *domain-independent* organisational package has been implemented in Java as per Fig. 3, Fig. 9 and other mechanisms introduced in this article. Most elements of the framework such as functional roles, players, composites and organisers have been implemented as standard abstract classes. The implementation of contracts is most interesting and is discussed in the following subsections. For the two *domain-specific* packages, the application programmer extends the abstract classes with domain-specific code in order to create an adaptive application. To test the framework, we have implemented a relatively closed application, i.e., the reference example used in this article [27]. The application simulates variable player performance and changing non-functional requirements in order to demonstrate the adaptability of the system.

6.2. Using association-aspects to implement contracts

Aspect-oriented methods and languages [24] seek to maintain the modularity of separate cross-cutting concerns in the design and source-code structures. Examples of cross-cutting concerns that have been modularised into aspects include security, logging, transaction management and the application of business rules. The AspectJ [15] extension to Java allows the programmer to define *pointcuts* that pick out certain *join points* (well-defined points in the program flow such as a *call* to a method). An *advice* is code that is executed when a join point that matches a pointcut is reached. *Aspects* encapsulate such pointcuts and advices. These units of modularity can model various cross-cutting concerns.

While AspectJ-like aspects have previously been used to add role behaviour to a single object [23], as far as we are aware they have not been used to implement associations between roles. Aspects, as they are currently implemented in AspectJ, do not easily represent the behavioural associations between objects [34]. Current implementations of AspectJ provide *per-object* aspects. These can be used to associate a unique aspect instance to either the executing object (*perthis*) or the target object (*pertarget*). When an advice execution is triggered in an object, the system looks up the aspect instance associated with that object and executes that instance. This allows the aspect to maintain a unique state for each object, but not for associations of groups of objects.

Sakurai et al. [33] developed *association-aspects* to enable an aspect *instance* to be associated with a group of objects. *Association-aspects* are implemented with a modification to the AspectJ. Association-aspects allow aspect *instances* to be created in the form

```
MyAssAspt a1 = new MyAssAspt (o1, o2, ... , oN);
```

where a1 is an aspect instance and o1…oN are a tuple of two or more objects associated with that instance. Association-aspects are declared with a *perobjects* modifier that takes as an argument a tuple of the associated objects. The ability to represent the associative state between objects in a group makes association-aspects suitable for representing contracts as we have defined them. Using our example above, the declaration of a concrete contract to bind two objects of type Foreman and ThingyMaker would be as follows:

```
public aspect FTContract extends SuperSub perobjects(Supervisor,
     Subordinate) {
private Foreman f;       //implements the Supervisor interface
private ThingyMaker t;   //implements the Subordinate interface
public FTContract(Foreman f, ThingyMaker t)    {
 associate(f, t); //creates the association-aspect
 this.f = f;
 this.t = t;
 ...
}
//contract term pointcut definitions and advice
 ...}
```

The `associate(f, t)` method that binds the objects in the association, is automatically defined when the `perobjects` modifier is used. The modifier also defines a `delete()` method that revokes an association. In contrast to per-object aspects in AspectJ, the creation and destruction of association-aspects instances is explicit.

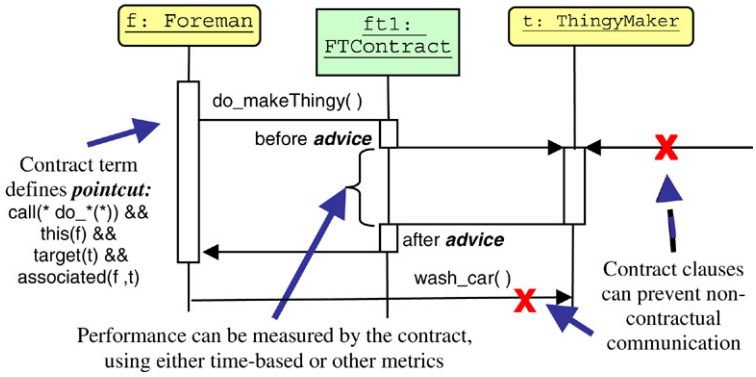

Fig. 12. Synchronous transaction between roles under contractual control.

The contract `FTContract` inherits from the abstract management contract `SuperSub`. This management contract defines Supervisor–Subordinate interaction patterns as defined in Table 2 (Section 4.3).

Once we have created a contract type in the above form, the creation and revocation of contract instances at run-time is straightforward. The following code would be invoked by an Organiser to create then delete a contract of type FTContract:

```
//create a contract between f1 and tm1
FTContract ft1 = new FTContract(f1, tm1);
//create a contract between f1 and tm2
FTContract ft2 = new FTContract(f1, tm2);
//revoke the contract between f1 and tm1
ft1.delete();
...
```

Fig. 12 schematically shows how an instance of ROAD contract (`ft1` of type `FTContract`) mediates a synchronous interaction between the two functional roles (`Foreman f` and `ThingyMaker t`). As described above, contracts have *terms* that define obligated transactions between the parties. The contract intercepts method calls between parties bound by the contract, where the signatures of those methods are defined in a contract term. In the case below, calls from `f` to `t` that start with a method name prefix `do_*` are intercepted. (Alternatively, methods could be *annotated* as in AspectJ 1.5, and a pointcut defined on the annotation.) Pointcuts can also be defined that prevent unauthorised method calls (any method call that is not specified in the contract) either between the parties to the contract, or from external entities as shown in Fig. 12. A more complete description of how ROAD contract terms are defined using pointcuts can be found in [10] and [14].

Advice is code in an aspect that is executed when a *pointcut* is reached in the execution flow. AspectJ supports a number of types of advice including *before* advice (executed just before the *join point* is reached) and *after* advice (executed after returning normally or after returning with an error). Rules for communication can be applied in a *before* advice. As illustrated above, contracts can make use of the change of state between *before* and *after* advice to measure performance of a contract clause. For example, the time elapsed between before and after advice can be used to calculate time-based metrics such as rate of production. Alternatively, some other utility function (such as cost) could be evaluated by accessing the execution context of the advice (either the state of the system or the environment).

Depending on the synchronisation method used (as described in Section 4.3) the interception points for measuring utility will vary. Fig. 12 shows a *synchronous* transaction between the parties (as in Fig. 7A). If the transaction is *asynchronous*, the post-transaction state is measured when a method call matching a response signature is intercepted (as in Fig. 7B).

Fig. 13 expands the three 'packages' schematised in Fig. 11 to show the contract-related classes and aspects, and shows how they are extend from the ROAD library. Because the abstract management contracts and roles described in this section are domain-independent, they provide the basis for a contract framework that has the potential to be reused in a number of domains.

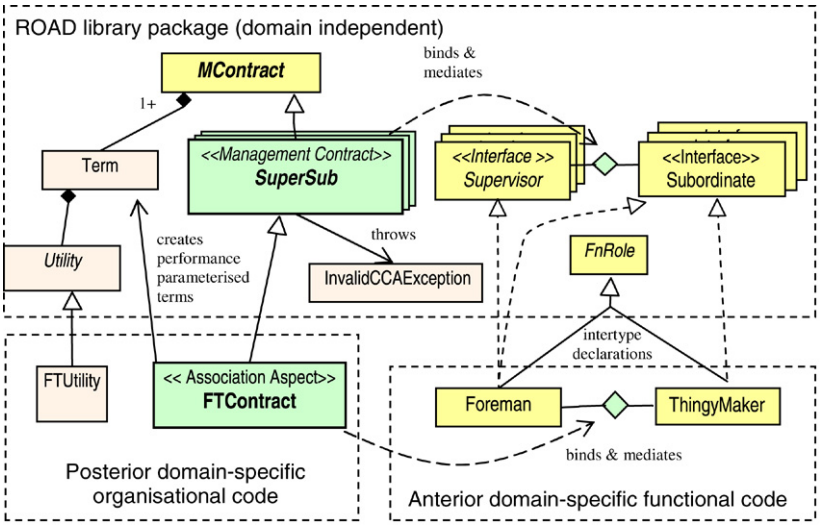

Fig. 13. Classes and aspects that define management and concrete ROAD contracts.

Contracts are implemented as an inheritance hierarchy of aspects (the bold classes in Fig. 13). The most general form of contract in this library (the `MContract` aspect) specifies general CCA definitions that can be used to define contract terms in the sub-aspects. The `MContract` aspect also provides mechanisms by which parties and clauses can be added to (and removed from) instances of contracts. Abstract management contracts (such as Supervisor–Subordinate, Peer–Peer, Auditor–Auditee etc.) inherit these functionalities from `MContract`, and define term transactions using CCA sequences as shown in Table 1. Such management contracts express various types of control relationships between roles by defining *contract clause pointcuts* as described above.

Concrete contracts, such as the `FTContract` in Fig. 13, inherit from a management contract type, and allow us to define performance characteristics for each term of a contract object. Domain-specific characteristics are passed as parameters to the Term class constructor when the clause is created. These parameters include: a reference to the contract that the term belongs to; the method signature, the direction of the invocation (`aToB` or `bToA`); the synchronisation type; a response signature if the type is asynchronous; and a domain-specific utility function object that defines the performance metrics. The abstract `Utility` class has a `calculatePerformance()` method that can be overridden by a concrete domain-specific utility sub-class (such as `FTUtility` in Fig. 13). Once the performance of a term is calculated, its state will be reported to the contract, if it is in breach or is underperforming. The contract, in turn, notifies its organiser of any underperformance.

In order for the functional-role classes to be able to work with management contracts, they need to implement the empty interfaces that represent any applicable operational-management roles such as `Supervisor` or `Subordinate`. They also need to extend the abstract functional role class `FnRole`. This abstract class provides some methods that are common to functional roles, such as keeping track of the contracts to which the role is a party. The creation of these dependencies does not require the alteration of the pre-existing functional classes but can be achieved by using an aspect with an *inter-type declaration*. Such declarations can create, at compile-time, the inheritance relationships and interfaces for functional roles. For instance, the following creates the inheritance relationships for the `ThingyMaker` and `DooverMaker` classes:

```
declare parents: {ThingyMaker || DooverMaker} extends FnRole
                                              implements Subordinate;
```

6.3. Discussion

Adding adaptive indirection to applications complicates the programming task. The ROAD approach requires that player and role interfaces are compatible, and the signatures of these interfaces are captured in the contracts that bind the roles. These interdependencies mean that coding roles, contracts and player interfaces as separate classes duplicates interface definitions and can potentially lead to inconsistencies. An underlying meta-model of these

interdependencies has been developed, and we are currently using this model as the basis for an Eclipse plug-in that will allow the developer to visually design organisational structures. Starting from the required and provided interface definitions of players, the programmer can select a subset of signatures from which to generate a role. Basic contract stubs are created by linking roles and matching the required and provided signatures. Alternatively, roles can be defined from scratch, and then the player interface stubs automatically generated. The aim of this on-going work is to reduce the complexity of programming adaptive applications, and to ensure consistency in the articulated structure.

While new contract *instances* can be added to an application at runtime, a limitation of the current implementation is that new contract *types* cannot be created on the fly. This prevents runtime *functional* (as opposed to non-functional) recomposition *within* a composite. This limitation results from the compile-time weaving of the contracts into the functional code. However, as composites are players new composites with different internal functional configurations can always be swapped at runtime.

Sakurai [33] has shown that there is little additional overhead in the use of *association-aspects* viz ordinary AspectJ aspects, thus there is little overhead in the basic interception mechanism of the contracts as they are currently implemented. However, it remains to be seen whether a domain-specific coordination system based on association-aspects would impose a significant run-time penalty — particularly if complex utility functions are used and need to be calculated with each interception.

7. Related work

ROAD extends work on role and associative modelling in [8,23,25,26]. Kendall [23] has shown how aspect-oriented approaches can be used to introduce role-behaviour to objects. Roles are encapsulated in aspects that are woven into the class structure. While these role-oriented approaches decouple the class structure, they do not explicitly define a coordination-system using management contracts. They are primarily concerned with role-object bindings rather than role associations.

The coordination model outlined here adopts a control-oriented [3] architectural approach, primarily focused on adaptivity rather than synchronisation. It has many similarities and some major differences with work by Andrade, Wermelinger and colleagues [1,2,36]. Both approaches represent contracts as first-class entities, and both use a layered architecture. In [1,36] the layers are Computation, Coordination and Configuration ('3C'). This is broadly similar to ROAD's four layer architecture (Computational-object, Functional-role, Management-contract, Organisation), 3C's Computation layer being similar to ROAD's Player and Functional role layers. 3C's contracts are method-centric rather than role-association-centric. They define a single interaction sequence that might involve many parties, whereas ROAD contracts are currently limited to two roles and may involve many types of interaction. Both approaches use contracts to model unstable aspects of the system, but 3C's focus is on business rules whereas ROAD focuses on performance variability. In 3C, there is no concept of a coordination network through which regulatory control messages pass.

PowerJava [5] extends the object-oriented paradigm and Java programming language with a pre-compiler to implement organisational roles. This approach has many similarities to ROAD in that institutions (composites) define roles that are played by players. A major difference in the approaches is that in powerJava institutions perform domain functions and maintain domain state themselves. Institutions give 'powers' to the object playing the roles, and there is no organiser role. In ROAD, on the other hand, all domain-function is implemented in the players. ROAD composites do not perform domain-functions by themselves, but rather are structured containers of contracted roles to which players are bound. In ROAD, the power to act within the composite is conferred by delegation from the composite to its roles, and via contracts between those roles, rather than having roles statically defined within an institution. Object Teams [20] can be used to define collaboration contexts (like ROAD and powerJava) but many such cross-cutting "teams" can be defined. In ROAD, on the other hand, organisations are disjoint (although a player may belong to more that one organisation). Like powerJava (and unlike ROAD), domain function can be split between a role and a player (base-object). Unlike ROAD and powerJava, Object Teams does not support adaptivity through indirection of instantiation; once a role-object is created the link to its base-object (player) cannot be changed.

This work also has much in common with adaptive architectural frameworks such as [9,18] in that these approaches define management entities that reconfigure a flexible component structure. However, these approaches do not separate role from implementation, and consequently cannot be recursively instantiated. Nor do the family of frameworks based on [18] effectively distribute the management function down through the structure.

The concept of a CCA in this paper is similar to the concept of a *communication act* in agent communication languages such as FIPA-ACL [17]. CCAs, as defined here, are far more limited in their extent. CCAs deal only with control communication, and do not have to take the intentionality of other parties into account [39]. Abstract communication act types have been used to control interaction in Web services Message Exchange Patterns (MEPs) in WSDL [35]. However, MEPs only express the direction of communication (IN or OUT) and whether or not the communication is robust or optional. In [6] a number of service interaction patterns are catalogued, including both bilateral and multilateral patterns. The bilateral patterns (e.g. "Relayed Request") may provide a catalogue from which transaction types expressed as CCA sequences could be developed. Abstract protocols also appear in component composition [37]. These are aimed at ensuring the compatibility of component interfaces and the sequencing of messages. They express the direction of the message and if it is a request or a response [31]. Control semantics (e.g. DO) are not captured.

Work on roles has also been undertaken in multi-agent systems (MAS) [22,30,38]. In particular, [39] extends the concept of a role model to an organisational model. While agents can be used as players within a ROAD framework they are constrained by the organisational structure. MAS systems, on the other hand, require all components to have deliberative capability and autonomy. These agents negotiate interactions with other agents to achieve system level goals. These negotiations occur within a more amorphous structure than is defined here.

8. Conclusion

Separating management concerns from functional concerns can make systems more adaptive. In this paper we have introduced a framework for creating coordination-systems that control the interactions between roles. These coordination systems can be developed separately and then superimposed onto functional entities. Such systems are built from a hierarchy of organiser roles that control the contracts between functional roles. ROAD contracts have abstract management and concrete functional levels. Abstract management contracts specify the type of communication acts that are permissible between the two parties, and they define the transaction patterns that can be measured for performance. Concrete contracts create the terms of the contract that specify, among other things, the performance obligations. Abstracting the management aspects of contracts makes possible, through contract inheritance, the reuse of authority patterns found in many types of organisational structure. The ability of organisers to make and break contracts between roles, and to control the binding between roles and player, provides a mechanism to reconfigure a system in a modular way so that it can better cope with changes in requirements, or changes in the environment.

There are a number of aspects of this framework that can be further developed. The set of CCAs that defined our example protocol could be extended and grounded in a more formal theory of control in organisations. Agent communication languages, such as [17], may provide the basis of a more rigorous definition. The concept of organisational regulation through indirect control is another area for future work. It follows that we need to develop some scheme of resource ownership or access rights. A further issue that needs addressing is the rules or mechanisms that need to be implemented when players are swapped between roles at runtime, particularly if there are long-lived transactions. We also assume that the interfaces of the functional roles are compatible if they are to enter into contracts. Issues of functional compatibility and component composition are active areas of research, and could be used to complement the ROAD framework.

References

[1] L. Andrade, J.L. Fiadeiro, J. Gouveia, G. Koutsoukos, Separating computation, coordination and configuration, Journal of Software Maintenance and Evolution: Research and Practice 14 (5) (2002) 353–369.

[2] L. Andrade, J.L. Fiadeiro, J. Gouveia, G. Koutsoukos, A. Lopes, M. Wermelinger, Patterns for coordination, in: Coordination'00, in: LNCS, vol. 1906, 2000.

[3] F. Arbab, What do you mean, coordination? Bulletin of the Dutch Association for Theoretical Computer Science (NVTI) (March) (1998).

[4] W.R. Ashby, An Introduction to Cybernetics, Chapman & Hall, London, 1956.

[5] M. Baldoni, G. Boella, L. van der Torre, Bridging agent theory and object orientation: Importing social roles in object oriented languages, in: Proceedings of PROMAS Workshop at AAMAS'05, 2005.

[6] A. Barros, M. Dumas, A. ter Hofstede, Service interaction patterns: towards a reference framework for service-based business process interconnection, Technical Report FIT-TR-2005-02, Faculty of Information Technology, QUT, Australia, 2005.

[7] A. Bracciali, A. Brogi, C. Canal, Dynamically adapting the behaviour of software components, in: Proceedings 5th International Conference on Coordination Models and Languages, Coordination'02, in: LNCS, vol. 2315, York, UK, 2002, pp. 88–95.
[8] D. Bäumer, D. Riehle, W. Siberski, M. Wulf, Pattern Languages of Program Design 4, Addison-Wesley, 2000, pp. 15–32.
[9] P. Collet, R. Rousseau, T. Coupaye, N. Rivierre, A contracting system for hierarchical components, in: SIGSOFT Symposium on Component-Based Software Engineering, CBSE'05, St-Louis, MO, USA, in: LNCS, vol. 3489, 2005, pp. 187–202.
[10] A. Colman, J. Han, Operational management contracts for adaptive software organisation, in: Proceedings of the Australian Software Engineering Conference, ASWEC 2005, Brisbane, Australia, 2005, pp. 170–179.
[11] A. Colman, J. Han, An organisational approach to building adaptive service-oriented systems, in: Proc. of 1st International Workshop on Engineering Service Compositions, WESC'05, IBM Research Report RC23821, Amsterdam, The Netherlands, 2005.
[12] A. Colman, J. Han, Organizational abstractions for adaptive systems, in: Proc. of the 38th Hawaii International Conference of System Sciences, Hawaii, USA, 2005.
[13] A. Colman, J. Han, Organizational roles and players, in: AAAI Fall Symposium, Roles, an Interdisciplinary Perspective, Arlington, VA, 2005, pp. 55–62.
[14] A. Colman, J. Han, Using associations aspects to implement organisational contracts, in: Proceedings of the 1st International Workshop on Coordination and Organisation, CoOrg 2005, Namur, Belgium, 2005.
[15] Eclipse Foundation, AspectJ, http://eclipse.org/aspectj/, 2004, (accessed 07.10.04).
[16] W. Emmerich, Engineering Distributed Objects, John Wiley & Sons, Chichester, 2000.
[17] The Foundation for Physical Intelligent Agents, FIPA Communicative Act Library Specification, http://www.fipa.org/specs/fipa00037/, 2002.
[18] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self-adaptation with reusable infrastructure, Computer 37 (10) (2004) 46–54.
[19] J. Han, K.K. Ker, Ensuring compatible interactions within component-based software systems, in: 10th Asia-Pacific Software Engineering Conference, APSEC 2003, Chiang Mai, Thailand, 2003, pp. 436–445.
[20] S. Herrmann, Object teams: Improving modularity for crosscutting collaborations, in: Net. Object Days 2002, Erfurt, Germany, 2002.
[21] E. Horvitz, Principles of mixed-initiative user interfaces, in: Proc. of ACM SIGCHI Conf. on Human Factors in Computing Systems, CHI 99, Pittsburgh, PA, USA, 1999.
[22] T. Juan, A. Pearce, L. Sterling, ROADMAP: Extending the Gaia methodology for complex open systems, in: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems, ACM, Bologna, Italy, 2002, pp. 3–10.
[23] E.A. Kendall, Role model designs and implementations with aspect-oriented programming, in: Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications, Denver, CO, 1999, pp. 353–369.
[24] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C.V. Lopes, C. Maeda, A. Mendhekar, Aspect-oriented programming, in: 11th European Conference on Object-Oriented Programming: ECOOP'97, Jyväskylä, Finland, 9–13 June 1997.
[25] B.B. Kristensen, K. Osterbye, Roles: Conceptual abstraction theory & practical language issues, in: Subjectivity in Object-Oriented Systems, Theory and Practice of Object Systems (TAPOS) (1996) (special issue).
[26] J.S. Lee, D.H. Bae, An enhanced role model for alleviating the role-binding anomaly, Software: Practice and Experience 32 (2002) 1317–1344.
[27] D.P. Linh, A. Colman, J. Han, The implementation of message synchronisation, queuing and allocation in the ROAD framework, Faculty of ICT, Swinburne University of Technology, Technical Report SUT.CeCSES-TR009, 2005.
[28] B. Meyer, Object-Oriented Software Construction, Prentice-Hall, New York, 1988.
[29] J. Odell, M. Nodine, R. Levy, A Metamodel for agents, roles, and groups, in: Agent-Oriented Software Engineering (AOSE) V, 2005.
[30] J. Odell, H.V.D. Parunak, S. Brueckner, J. Sauter, Changing roles: Dynamic role assignment, Journal of Object Technology, ETH Zurich 2 (5) (2003) 77–86.
[31] F. Plasil, S. Visnovsky, Behavior protocols for software components, IEEE Transactions on Software Engineering 28 (11) (2002) 1056–1076.
[32] D. Riehle, Bureaucracy, in: R. Martin, et al. (Eds.), Pattern Languages of Program Design 3, Addison-Wesley, Reading, MA, 1998, pp. 163–186.
[33] K. Sakurai, H. Masuharat, N. Ubayashi, S. Matsuura, S. Komiya, Association aspects, in: Proc. of the AOSD'04, Lancaster, UK, 2004, pp. 16–25.
[34] K. Sullivan, L. Gu, Y. Cai, Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for *AspectJ*, in: Proc. of the 1st International Conference on AOSD 02, Enschede, The Netherlands, 2002, pp. 19–26.
[35] W3C, Web Services Description Language (WSDL) Version 2.0 Primer, W3C Working Draft, 10 May 2005, http://www.w3.org/TR/wsdl20-primer/, 2005.
[36] M. Wermelinger, J.L. Fiadeiro, L. Andrade, G. Koutsoukos, J. Gouveia, Separation of core concerns: Computation, coordination, and configuration, in: Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay, FL, 2001.
[37] D.M. Yellin, R.E. Strom, Protocol specifications and component adaptors, ACM Transactions on Programming Languages and Systems 19 (2) (1997) 292–333.
[38] F. Zambonelli, N.R. Jennings, M.J. Wooldridge, Organisational abstractions for the analysis and design of multi-agent systems, in: Workshop on Agent-oriented Software Engineering, ICSE 2000, 2000.
[39] F. Zambonelli, N.R. Jennings, M. Wooldridge, Developing multiagent systems: The Gaia methodology, ACM Transactions on Software Engineering and Methodology 12 (3) (2003) 317–370.